FROM POSTGRESQL TO SQLITE IN RAILS

MIGRATION JOURNEY, CHALLENGES, AND LASTING TRADE-OFFS

By Wojtek Wrona @wojtodzio
X, Bluesky, LinkedIn

MOTIVATION

MOTIVATION

- Team lead at a small startup

MOTIVATION

- Team lead at a small startup
- 8 people total, 3 backend developers

MOTIVATION

- Team lead at a small startup
- 8 people total, 3 backend developers
- Rails app: ~80k LOC + ~160k tests

MOTIVATION

- Team lead at a small startup
- 8 people total, 3 backend developers
- Rails app: ~80k LOC + ~160k tests
- Tight budget, tight deadlines, no time for optimizations

MOTIVATION

- Team lead at a small startup
- 8 people total, 3 backend developers
- Rails app: ~80k LOC + ~160k tests
- Tight budget, tight deadlines, no time for optimizations
- No dedicated DevOps team

MOTIVATION

- Team lead at a small startup
- 8 people total, 3 backend developers
- Rails app: ~80k LOC + ~160k tests
- Tight budget, tight deadlines, no time for optimizations
- No dedicated DevOps team
- Inspired by last year Stephen’s wroclove.rb talk

MOTIVATION

- Team lead at a small startup
- 8 people total, 3 backend developers
- Rails app: ~80k LOC + ~160k tests
- Tight budget, tight deadlines, no time for optimizations
- No dedicated DevOps team
- Inspired by last year Stephen’s wroclove.rb talk
- Rails core support for SQLite improved significantly

MOTIVATION

- Team lead at a small startup
- 8 people total, 3 backend developers
- Rails app: ~80k LOC + ~160k tests
- Tight budget, tight deadlines, no time for optimizations
- No dedicated DevOps team
- Inspired by last year Stephen’s wroclove.rb talk
- Rails core support for SQLite improved significantly
- SolidCache, SolidQueue, SolidCable

WHY MIGRATE?

WHY MIGRATE?

- Embedded DB, no extra services (Postgres/Redis)

WHY MIGRATE?

- Embedded DB, no extra services (Postgres/Redis)
- Zero latency, no need to worry about N+1: less time spent on optimization

WHY MIGRATE?

- Embedded DB, no extra services (Postgres/Redis)
- Zero latency, no need to worry about N+1: less time spent on optimization
- Wast majority of our queries are reads

WHY MIGRATE?

- Embedded DB, no extra services (Postgres/Redis)
- Zero latency, no need to worry about N+1: less time spent on optimization
- Wast majority of our queries are reads
- Easy E2E Testing (DB per test run)

WHY MIGRATE?

- Embedded DB, no extra services (Postgres/Redis)
- Zero latency, no need to worry about N+1: less time spent on optimization
- Wast majority of our queries are reads
- Easy E2E Testing (DB per test run)
- bundle is enough -> removed Docker

WHY MIGRATE?

- Embedded DB, no extra services (Postgres/Redis)
- Zero latency, no need to worry about N+1: less time spent on optimization
- Wast majority of our queries are reads
- Easy E2E Testing (DB per test run)
- bundle is enough -> removed Docker
- Single server, easier deployment

WHY MIGRATE?

- Embedded DB, no extra services (Postgres/Redis)
- Zero latency, no need to worry about N+1: less time spent on optimization
- Wast majority of our queries are reads
- Easy E2E Testing (DB per test run)
- bundle is enough -> removed Docker
- Single server, easier deployment
- Potential for significant cost saving

COSTS

Platform	DB	Server (16vCPU total)	Redis	User	Total
Render - many small instances	$100 (2CPU)	$25 * 14 1vCPU Standard + $85 2vCPU Worker	$32 Standard	$19 * 4	~$643
Render - biggest instances	$100 (2CPU)	$450 * 2 8vCPU	$32 Standard	$19 * 4	~$1 108
Hetzner + Hatchbox	$0	$67	$0	$10 (Hatchbox)	~$77

RESULTS

RESULTS

- Migration completed smoothly (approx. 3 dev days total)

RESULTS

- Migration completed smoothly (approx. 3 dev days total)
- Requests ~40-50% faster (also moved infra)
  - Migrated in January '25, DB weighs 1.7GB, has 187 tables, the largest table has 2.8 million rows

RESULTS

- Migration completed smoothly (approx. 3 dev days total)
- Requests ~40-50% faster (also moved infra)
  - Migrated in January '25, DB weighs 1.7GB, has 187 tables, the largest table has 2.8 million rows
- Infrastructure costs slashed

RESULTS

- Migration completed smoothly (approx. 3 dev days total)
- Requests ~40-50% faster (also moved infra)
  - Migrated in January '25, DB weighs 1.7GB, has 187 tables, the largest table has 2.8 million rows
- Infrastructure costs slashed
- DevOps overhead significantly reduced

RESULTS

- Migration completed smoothly (approx. 3 dev days total)
- Requests ~40-50% faster (also moved infra)
  - Migrated in January '25, DB weighs 1.7GB, has 187 tables, the largest table has 2.8 million rows
- Infrastructure costs slashed
- DevOps overhead significantly reduced
- More team focus on features, less on infra/optimization

RESULTS

- Migration completed smoothly (approx. 3 dev days total)
- Requests ~40-50% faster (also moved infra)
  - Migrated in January '25, DB weighs 1.7GB, has 187 tables, the largest table has 2.8 million rows
- Infrastructure costs slashed
- DevOps overhead significantly reduced
- More team focus on features, less on infra/optimization
- Almost free backups with point-in-time recovery (Litestream)

MIGRATION STRATEGY: INCREMENTAL & REVERSIBLE

MIGRATION STRATEGY: INCREMENTAL & REVERSIBLE

- Goal: Migrate without a "big bang" cutover

MIGRATION STRATEGY: INCREMENTAL & REVERSIBLE

- Goal: Migrate without a "big bang" cutover
- Approach:

MIGRATION STRATEGY: INCREMENTAL & REVERSIBLE

- Goal: Migrate without a "big bang" cutover
- Approach:
  - Prepare the app while still on PostgreSQL.

MIGRATION STRATEGY: INCREMENTAL & REVERSIBLE

- Goal: Migrate without a "big bang" cutover
- Approach:
  - Prepare the app while still on PostgreSQL.
  - Migrate schema/types incrementally. Ensure every step is reversible & testable.

MIGRATION STRATEGY: INCREMENTAL & REVERSIBLE

- Goal: Migrate without a "big bang" cutover
- Approach:
  - Prepare the app while still on PostgreSQL.
  - Migrate schema/types incrementally. Ensure every step is reversible & testable.
  - Only switch the database adapter and migrate data after preparation.

MIGRATION STRATEGY: INCREMENTAL & REVERSIBLE

- Goal: Migrate without a "big bang" cutover
- Approach:
  - Prepare the app while still on PostgreSQL.
  - Migrate schema/types incrementally. Ensure every step is reversible & testable.
  - Only switch the database adapter and migrate data after preparation.
  - Fix all the failing tests.

STEP 1: HANDLING UNSUPPORTED TYPES (STILL ON PG)

STEP 1: HANDLING UNSUPPORTED TYPES (STILL ON PG)

- Challenge: PostgreSQL has types SQLite doesn't (e.g., inet, uuid, interval, array)

STEP 1: HANDLING UNSUPPORTED TYPES (STILL ON PG)

- Challenge: PostgreSQL has types SQLite doesn't (e.g., inet, uuid, interval, array)
- SQLite: Uses dynamic typing (stores TEXT, INTEGER, REAL, BLOB, NULL).

STEP 1: HANDLING UNSUPPORTED TYPES (STILL ON PG)

- Challenge: PostgreSQL has types SQLite doesn't (e.g., inet, uuid, interval, array)
- SQLite: Uses dynamic typing (stores TEXT, INTEGER, REAL, BLOB, NULL).
- Problem: How to store PG-specific data in SQLite's basic types without breaking the app?

SOLUTION: CUSTOM ACTIVERECORD TYPES

SOLUTION: CUSTOM ACTIVERECORD TYPES

- Rails allows defining custom types (ActiveRecord::Type).

SOLUTION: CUSTOM ACTIVERECORD TYPES

- Rails allows defining custom types (ActiveRecord::Type).
- We can tell Rails how to serialize Ruby objects to SQLite-compatible types (like String/TEXT) and how to deserialize them back into the expected Ruby objects.

SOLUTION: CUSTOM ACTIVERECORD TYPES

- Rails allows defining custom types (ActiveRecord::Type).
- We can tell Rails how to serialize Ruby objects to SQLite-compatible types (like String/TEXT) and how to deserialize them back into the expected Ruby objects.
- Application code (models, controllers) remains unchanged!

EXAMPLE: `inet` -> `string` (1/3 - CUSTOM TYPE)

```ruby
class IpAddressType < ActiveRecord::Type::String
  def serialize(value)
    return if value.nil?

    case value
    when IPAddr
      "#{value}/#{value.prefix}"
    when String
      ip_addr = IPAddr.new(value)
      "#{ip_addr}/#{ip_addr.prefix}"
    else
      raise ArgumentError, "Invalid IP address: #{value}"
    end
  end

  def cast_value(value)
    # ...
  end
end
```

EXAMPLE: `inet` -> `string` (1/3 - CUSTOM TYPE)

```ruby
class IpAddressType < ActiveRecord::Type::String
  def serialize(value)
    # ...
  end

  def cast_value(value)
    case value
    when IPAddr
      value
    when String
      IPAddr.new(value)
    else
      raise ArgumentError, "Invalid IP address: #{value}"
    end
  end
end
```

EXAMPLE: `inet` -> `string` (2/3 - MODEL)

```ruby
class User < ApplicationRecord
  attribute :current_sign_in_ip, IpAddressType.new
  attribute :last_sign_in_ip, IpAddressType.new
end
```

Rails now uses `IpAddressType` for serialization/deserialization.

EXAMPLE: `inet` -> `string` (3/3 - THE MIGRATION)

```ruby
class MigrateInetToString < ActiveRecord::Migration[7.0]
  def up
    change_column :users, :current_sign_in_ip, :string
    change_column :users, :last_sign_in_ip, :string
    # ... other inet columns
  end

  def down
    change_column :users, :current_sign_in_ip, :inet, using: 'curr
    change_column :users, :last_sign_in_ip, :inet, using: 'last_si
    # ... other inet columns
  end
end
```

- **Key:** This runs on PostgreSQL!
- Fully reversible. Deploy, test, revert if needed.

EXAMPLE: `inet` -> `string` (3/3 - THE MIGRATION)

```ruby
class MigrateInetToString < ActiveRecord::Migration[7.0]
  def up
    change_column :users, :current_sign_in_ip, :string
    change_column :users, :last_sign_in_ip, :string
    # ... other inet columns
  end

  def down
    change_column :users, :current_sign_in_ip, :inet, using: 'curr
    change_column :users, :last_sign_in_ip, :inet, using: 'last_si
    # ... other inet columns
  end
end
```

- **Key:** This runs on PostgreSQL!
- Fully reversible. Deploy, test, revert if needed.

CHALLENGE: ARRAYS

CHALLENGE: ARRAYS

- Arrays are trickier. We used them quite a bit.

CHALLENGE: ARRAYS

- Arrays are trickier. We used them quite a bit.
- Serializing to string loses query capabilities.

CHALLENGE: ARRAYS

- Arrays are trickier. We used them quite a bit.
- Serializing to string loses query capabilities.
- Solution: Leverage SQLite's native JSON support.

GEM FOR CUSTOM TYPES

The common types we created:
https://github.com/wojtodzio/activerecord-sqlite-types

SOLID STACK

SOLID STACK

- SolidCache, SolidQueue, SolidCable - you don't need Redis anymore.

```
gem 'solid_cache'
gem 'solid_queue'
gem 'solid_cable'
```

SOLID STACK

- SolidCache, SolidQueue, SolidCable - you don't need Redis anymore.

```
gem 'solid_cache'
gem 'solid_queue'
gem 'solid_cable'
```

- Each uses its own separate SQLite database file.

SOLID STACK

- SolidCache, SolidQueue, SolidCable - you don't need Redis anymore.

```
gem 'solid_cache'
gem 'solid_queue'
gem 'solid_cable'
```

- Each uses its own separate SQLite database file.
- Tested independently on staging.

CHALLENGE: DB CONSTRAINTS

CHALLENGE: DB CONSTRAINTS

- We use CHECK constraints extensively.

CHALLENGE: DB CONSTRAINTS

- We use CHECK constraints extensively.
- PostgreSQL functions (now(), casting ::integer) differ from SQLite (CURRENT_TIMESTAMP, CAST(... AS INTEGER)).

CHALLENGE: DB CONSTRAINTS

- We use CHECK constraints extensively.
- PostgreSQL functions (now(), casting ::integer) differ from SQLite (CURRENT_TIMESTAMP, CAST(... AS INTEGER)).
- Constraints written for PG fail in SQLite, and vice-versa.

SOLUTION: TWO-STEP CONSTRAINT MIGRATION

```ruby
class DropConstraintBeforeSqlite < ActiveRecord::Migration[8.0]
  def change
    remove_check_constraint(
      :automated_notifications,
      <<-SQL.squish,
        ((template_name IS NOT NULL)::integer +
        (sms_body IS NOT NULL)::integer +
        (push_notification_body IS NOT NULL)::integer) > 0
      SQL
      name: 'template_name_' \
        'or_sms_body_or_' \
        'push_notification_body_present'
    )

    remove_check_constraint(
      :line_items,
      <<-SQL.squish,
        ((email IS NOT NULL)::integer +
        (phone_number IS NOT NULL)::integer) = 1
      SQL
      name: 'only_email_or_phone_number'
    )
  end
end
```

SOLUTION: TWO-STEP CONSTRAINT MIGRATION

```ruby
class AddConstraintsBackInSqlite < ActiveRecord::Migration[8.1]
  def change
    raise if ActiveRecord::Base.connection.adapter_name.downcase != 'sqlite'

    add_check_constraint(
      :automated_notifications,
      <<-SQL.squish,
        template_name IS NOT NULL OR sms_body IS NOT NULL
        OR push_notification_body IS NOT NULL
      SQL
      name: 'template_name_' \
        'or_sms_body_or_' \
        'push_notification_body_present'
    )

    add_check_constraint(
      :line_items,
      <<-SQL.squish,
        CAST(email IS NOT NULL AS INTEGER) +
        CAST(phone_number IS NOT NULL AS INTEGER) = 1
      SQL
      name: 'only_email_or_phone_number'
    )
  end
end
```

STEP 2: THE ACTUAL DATA MIGRATION

STEP 2: THE ACTUAL DATA MIGRATION

Types migrated, constraints dropped: we don't have anything in the schema that SQLite couldn't handle

STEP 2: THE ACTUAL DATA MIGRATION

Types migrated, constraints dropped: we don't have anything in the schema that SQLite couldn't handle

1. Point your database.yml to a new SQLite file

STEP 2: THE ACTUAL DATA MIGRATION

Types migrated, constraints dropped: we don't have anything in the schema that SQLite couldn't handle

1. Point your database.yml to a new SQLite file
2. Create a new database

STEP 2: THE ACTUAL DATA MIGRATION

Types migrated, constraints dropped: we don't have anything in the schema that SQLite couldn't handle

1. Point your database.yml to a new SQLite file
2. Create a new database
3. Run github.com/hirefrank/pg-to-sqlite

STEP 2: THE ACTUAL DATA MIGRATION

Types migrated, constraints dropped: we don't have anything in the schema that SQLite couldn't handle

1. Point your database.yml to a new SQLite file
2. Create a new database
3. Run github.com/hirefrank/pg-to-sqlite
4. Ran the test suite!

SQLITE EXTENSIONS (`sqlean`)

- SQLite core is lite; extensions add functionality.
- `sqlean` gem bundles many useful C extensions:
  - Crypto, Define, FileIO, Fuzzy, IPAddr, Math, Regexp, Stats, Text, Unicode, UUID, VSV

```
gem 'sqlean'
```

```
primary: &primary
  <<: *default
  database: storage/<%= Rails.env %>.sqlite3
  extensions:
    - SQLean::UUID
    - SQLean::Text
    - SQLean::Regexp
```

CHALLENGE: CASE-INSENSITIVE SEARCH (`ILIKE`)

CHALLENGE: CASE-INSENSITIVE SEARCH (`ILIKE`)

- Big Surprise: SQLite has no ILIKE.

CHALLENGE: CASE-INSENSITIVE SEARCH (`ILIKE`)

- Big Surprise: SQLite has no ILIKE.
- LIKE is case-insensitive only for ASCII by default.

CHALLENGE: CASE-INSENSITIVE SEARCH (`ILIKE`)

- Big Surprise: SQLite has no ILIKE.
- LIKE is case-insensitive only for ASCII by default.
- "The reason for this is that doing full Unicode case-insensitive comparisons and case conversions requires tables and logic that would nearly double the size of the SQLite library."

CHALLENGE: CASE-INSENSITIVE SEARCH (`ILIKE`)

- Big Surprise: SQLite has no ILIKE.
- LIKE is case-insensitive only for ASCII by default.
- "The reason for this is that doing full Unicode case-insensitive comparisons and case conversions requires tables and logic that would nearly double the size of the SQLite library."
- sqlean/text provides TEXT_LOWER(), TEXT_UPPER(), TEXT_LIKE() etc. You can alias them to override built-in functions

OUR SOLUTION: COLLATE NOCASE & VIRTUAL COLUMNS

OUR SOLUTION: `COLLATE NOCASE` & VIRTUAL COLUMNS

Always Insensitive Columns (e.g., email):

OUR SOLUTION: COLLATE NOCASE & VIRTUAL COLUMNS

Always Insensitive Columns (e.g., email):

```
change_column :users, :email, :string, collation: 'text_nocase'
```

OUR SOLUTION: COLLATE NOCASE & VIRTUAL COLUMNS

Always Insensitive Columns (e.g., email):

```
change_column :users, :email, :string, collation: 'text_nocase'
```

Sometimes Insensitive Columns (e.g., name search):.

OUR SOLUTION: COLLATE NOCASE & VIRTUAL COLUMNS

Always Insensitive Columns (e.g., email):

```
change_column :users, :email, :string, collation: 'text_nocase'
```

Sometimes Insensitive Columns (e.g., name search):.

```
1 add_column :user_profiles,
2    :lower_full_name,
3    :virtual,
4    type: :string,
5    as: "TEXT_LOWER(first_name) || ' ' || TEXT_LOWER(last_name)",
6    stored: true
7 add_index :profiles, :lower_full_name
```

OUR SOLUTION: COLLATE NOCASE & VIRTUAL COLUMNS

Always Insensitive Columns (e.g., email):

```
change_column :users, :email, :string, collation: 'text_nocase'
```

Sometimes Insensitive Columns (e.g., name search):.

```
1 add_column :user_profiles,
2    :lower_full_name,
3    :virtual,
4    type: :string,
5    as: "TEXT_LOWER(first_name) || ' ' || TEXT_LOWER(last_name)",
6    stored: true
7 add_index :profiles, :lower_full_name
```

CASE-INSENSITIVE SEARCH: EXAMPLES

CASE-INSENSITIVE SEARCH: EXAMPLES

Always Insensitive (`COLLATE NOCASE`):

CASE-INSENSITIVE SEARCH: EXAMPLES

Always Insensitive (`COLLATE NOCASE`):

```
# Works case-insensitively automatically (incl. unique constraints)
User.find_by(email: 'User@Example.COM')
```

CASE-INSENSITIVE SEARCH: EXAMPLES

Always Insensitive (`COLLATE NOCASE`):

```ruby
# Works case-insensitively automatically (incl. unique constraints)
User.find_by(email: 'User@Example.COM')
```

Sometimes Insensitive (Virtual Column):

CASE-INSENSITIVE SEARCH: EXAMPLES

Always Insensitive (`COLLATE NOCASE`):

```ruby
# Works case-insensitively automatically (incl. unique constraints)
User.find_by(email: 'User@Example.COM')
```

Sometimes Insensitive (Virtual Column):

```ruby
# PostgreSQL
Profile.where("full_name ILIKE ?", "%#{term}%")

# SQLite (using virtual column)
Profile.where("lower_full_name LIKE ?", "%#{term.downcase}%")
# Or using GLOB (similar to LIKE but uses
# the Unix file globbing syntax for its wildcards)
Profile.where("lower_full_name GLOB ?", "*#{term.downcase}*")
```

CASE-INSENSITIVE SEARCH: EXAMPLES

Always Insensitive (`COLLATE NOCASE`):

```ruby
# Works case-insensitively automatically (incl. unique constraints)
User.find_by(email: 'User@Example.COM')
```

Sometimes Insensitive (Virtual Column):

```ruby
# PostgreSQL
Profile.where("full_name ILIKE ?", "%#{term}%")

# SQLite (using virtual column)
Profile.where("lower_full_name LIKE ?", "%#{term.downcase}%")
# Or using GLOB (similar to LIKE but uses
# the Unix file globbing syntax for its wildcards)
Profile.where("lower_full_name GLOB ?", "*#{term.downcase}*")
```

CASE-INSENSITIVE SEARCH: EXAMPLES

Always Insensitive (`COLLATE NOCASE`):

```ruby
# Works case-insensitively automatically (incl. unique constraints)
User.find_by(email: 'User@Example.COM')
```

Sometimes Insensitive (Virtual Column):

```ruby
# PostgreSQL
Profile.where("full_name ILIKE ?", "%#{term}%")

# SQLite (using virtual column)
Profile.where("lower_full_name LIKE ?", "%#{term.downcase}%")
# Or using GLOB (similar to LIKE but uses
# the Unix file globbing syntax for its wildcards)
Profile.where("lower_full_name GLOB ?", "*#{term.downcase}*")
```

MISSING FUNCTIONS: ARRAY QUERYING

MISSING FUNCTIONS: ARRAY QUERYING

- Remember array columns are now json? PG array functions (?|, &&, etc.) don't exist.

MISSING FUNCTIONS: ARRAY QUERYING

- Remember array columns are now json? PG array functions (?|, &&, etc.) don't exist.
- Solution 1: install array extension

MISSING FUNCTIONS: ARRAY QUERYING

- Remember array columns are now json? PG array functions (?|, &&, etc.) don't exist.
- Solution 1: install array extension
- Solution 2: Query JSON: Use SQLite's JSON functions (json_each, json_extract).

MISSING FUNCTIONS: ARRAY QUERYING

Abstract JSON querying (based on Stephen's article: https://fractaledmind.github.io/2023/09/12/enhancing-rails-sqlite-array-columns/).

```ruby
# PostgreSQL Array Query
User.where("personality_traits ?| ARRAY[:traits]", traits:)

# SQLite JSON Query
User.with_any_personality_traits(*traits)
```

POWER FEATURE: CUSTOM RUBY FUNCTIONS!

POWER FEATURE: CUSTOM RUBY FUNCTIONS!

- SQLite is embedded -> SQL can call your Ruby code!

POWER FEATURE: CUSTOM RUBY FUNCTIONS!

- SQLite is embedded -> SQL can call your Ruby code!
- Define custom SQL functions/aggregates using Ruby blocks.

RUBY SCALAR FUNCTIONS

```ruby
db = ApplicationRecord.connection.raw_connection
db.create_function("SORT_LETTERS", 1) do |func, value|
  if value.nil?
    func.result = nil
  else
    func.result = value.split('').sort.join('')
  end
end
UserProfile.limit(2).pluck('first_name')
=> ["John", "Joe"]
UserProfile.limit(2).pluck('SORT_LETTERS(first_name)')
=> ["Jhno", "Jeo"]
```

RUBY SCALAR FUNCTIONS

```ruby
db = ApplicationRecord.connection.raw_connection
db.create_function("SORT_LETTERS", 1) do |func, value|
  if value.nil?
    func.result = nil
  else
    func.result = value.split('').sort.join('')
  end
end
UserProfile.limit(2).pluck('first_name')
=> ["John", "Joe"]
UserProfile.limit(2).pluck('SORT_LETTERS(first_name)')
=> ["Jhno", "Jeo"]
```

CUSTOM AGGREGATE FUNCTIONS

```ruby
db = ApplicationRecord.connection.raw_connection
db.create_aggregate("COUNT_LETTER", 2) do
  step do |func, value, letter|
    func[:total] ||= 0
    func[:total] += value.count(letter) if value.present?
  end

  finalize do |func|
    func.result = func[:total] || 0
  end
end
UserProfile.pluck(Arel.sql("COUNT_LETTER(first_name, 'a')"))
=> [9553]
Event.joins(:users).group(:id)
     .having("COUNT_LETTER(users.email, 'a') > 257")
     .pluck(:id)
=> [159]
```

CUSTOM AGGREGATE FUNCTIONS

```ruby
db = ApplicationRecord.connection.raw_connection
db.create_aggregate("COUNT_LETTER", 2) do
  step do |func, value, letter|
    func[:total] ||= 0
    func[:total] += value.count(letter) if value.present?
  end

  finalize do |func|
    func.result = func[:total] || 0
  end
end
UserProfile.pluck(Arel.sql("COUNT_LETTER(first_name, 'a')"))
=> [9553]
Event.joins(:users).group(:id)
     .having("COUNT_LETTER(users.email, 'a') > 257")
     .pluck(:id)
=> [159]
```

CUSTOM AGGREGATE FUNCTIONS

```ruby
db = ApplicationRecord.connection.raw_connection
db.create_aggregate("COUNT_LETTER", 2) do
  step do |func, value, letter|
    func[:total] ||= 0
    func[:total] += value.count(letter) if value.present?
  end

  finalize do |func|
    func.result = func[:total] || 0
  end
end
UserProfile.pluck(Arel.sql("COUNT_LETTER(first_name, 'a')"))
=> [9553]
Event.joins(:users).group(:id)
     .having("COUNT_LETTER(users.email, 'a') > 257")
     .pluck(:id)
=> [159]
```

DIFFERENCE: NULL ORDERING

- In PostgreSQL: NULL is the highest possible value
- In SQLite: NULL is the lowest possible value

Solution: Explicitly use `NULLS FIRST` or `NULLS LAST` (ANSI SQL).

```
Model.order(Model.arel_table[:column].asc.nulls_last)
Model.order(Model.arel_table[:column].desc.nulls_first)
```

TRADE-OFF: CONCURRENCY & LOCKING

TRADE-OFF: CONCURRENCY & LOCKING

- PostgreSQL: Sophisticated row-level locking. Multiple writers can often proceed if touching different rows.

TRADE-OFF: CONCURRENCY & LOCKING

- PostgreSQL: Sophisticated row-level locking. Multiple writers can often proceed if touching different rows.
- SQLite (with WAL mode - default in new Rails): Allows multiple readers concurrently with one writer.

RAILS & BEGIN IMMEDIATE

RAILS & `BEGIN IMMEDIATE`

- Rails now uses BEGIN IMMEDIATE TRANSACTION with SQLite.

RAILS & BEGIN IMMEDIATE

- Rails now uses BEGIN IMMEDIATE TRANSACTION with SQLite.
- This acquires a "RESERVED" lock immediately at the start of the transaction. Effectively prevents any other write to the database until the transaction commits or rolls back.

RAILS & BEGIN IMMEDIATE

- Rails now uses BEGIN IMMEDIATE TRANSACTION with SQLite.
- This acquires a "RESERVED" lock immediately at the start of the transaction. Effectively prevents any other write to the database until the transaction commits or rolls back.
- Fine for short, typical Rails model updates (usually extremely fast in SQLite!).

RAILS & BEGIN IMMEDIATE

- Rails now uses BEGIN IMMEDIATE TRANSACTION with SQLite.
- This acquires a "RESERVED" lock immediately at the start of the transaction. Effectively prevents any other write to the database until the transaction commits or rolls back.
- Fine for short, typical Rails model updates (usually extremely fast in SQLite!).
- BUT... what about longer transactions?

PROBLEM: LONG TRANSACTIONS

```ruby
ApplicationRecord.transaction do
  # DB write (acquires IMMEDIATE lock)
  newsletter_subscription.update!(archived_at: Time.current)

  # Slow external network call
  MailchimpWrapper.archive(newsletter_subscription.email)

  # Transaction commits/rolls back (releases lock)
end
```

PROBLEM: LONG TRANSACTIONS

```ruby
ApplicationRecord.transaction do
  # DB write (acquires IMMEDIATE lock)
  newsletter_subscription.update!(archived_at: Time.current)

  # Slow external network call
  MailchimpWrapper.archive(newsletter_subscription.email)

  # Transaction commits/rolls back (releases lock)
end
```

PROBLEM: LONG TRANSACTIONS

```ruby
ApplicationRecord.transaction do
  # DB write (acquires IMMEDIATE lock)
  newsletter_subscription.update!(archived_at: Time.current)

  # Slow external network call
  MailchimpWrapper.archive(newsletter_subscription.email)

  # Transaction commits/rolls back (releases lock)
end
```

PROBLEM: LONG TRANSACTIONS

```ruby
ApplicationRecord.transaction do
  # DB write (acquires IMMEDIATE lock)
  newsletter_subscription.update!(archived_at: Time.current)

  # Slow external network call
  MailchimpWrapper.archive(newsletter_subscription.email)

  # Transaction commits/rolls back (releases lock)
end
```

PROBLEM: LONG TRANSACTIONS

```ruby
ApplicationRecord.transaction do
  # DB write (acquires IMMEDIATE lock)
  newsletter_subscription.update!(archived_at: Time.current)

  # Slow external network call
  MailchimpWrapper.archive(newsletter_subscription.email)

  # Transaction commits/rolls back (releases lock)
end
```

- In PG: Only locks the newsletter_subscription row (mostly).

PROBLEM: LONG TRANSACTIONS

```ruby
ApplicationRecord.transaction do
  # DB write (acquires IMMEDIATE lock)
  newsletter_subscription.update!(archived_at: Time.current)

  # Slow external network call
  MailchimpWrapper.archive(newsletter_subscription.email)

  # Transaction commits/rolls back (releases lock)
end
```

- In PG: Only locks the newsletter_subscription row (mostly).
- In SQLite: Locks the entire database for writes for the duration of the MailchimpWrapper call!

SOLUTION: AVOID TRANSACTIONS AROUND SLOW OPERATIONS

Refactor to perform the slow operation *outside* the transaction.

```
# Perform slow operation first
MailchimpWrapper.archive(newsletter_subscription.email)

# Only update DB if external call succeeded
# Short transaction, minimal lock duration
newsletter_subscription.update!(archived_at: Time.current)
```

SOLUTION: SEPARATE DBS

SOLUTION: SEPARATE DBS

In models:

SOLUTION: SEPARATE DBS

In models:

```ruby
class AnalyticsRecord < ActiveRecord::Base
  self.abstract_class = true
  connects_to database: { writing: :analytics, reading: :analytics
end
```

SOLUTION: SEPARATE DBS

In models:

```ruby
class AnalyticsRecord < ActiveRecord::Base
  self.abstract_class = true
  connects_to database: { writing: :analytics, reading: :analytics
end
```

SOLUTION: SEPARATE DBS

In models:

```ruby
class AnalyticsRecord < ActiveRecord::Base
  self.abstract_class = true
  connects_to database: { writing: :analytics, reading: :analytics
end
```

In config/database.yml:

SOLUTION: SEPARATE DBS

In models:

```ruby
class AnalyticsRecord < ActiveRecord::Base
  self.abstract_class = true
  connects_to database: { writing: :analytics, reading: :analytics
end
```

In config/database.yml:

```yaml
analytics:
  <<: *default
  migrations_paths: db/analytics_migrate
  database: storage/<%= Rails.env %>-analytics.sqlite3
```

SOLUTION: SEPARATE DBS

In models:

```ruby
class AnalyticsRecord < ActiveRecord::Base
  self.abstract_class = true
  connects_to database: { writing: :analytics, reading: :analytics
end
```

In config/database.yml:

```yaml
analytics:
  <<: *default
  migrations_paths: db/analytics_migrate
  database: storage/<%= Rails.env %>-analytics.sqlite3
```

SOLUTION: SEPARATE DBS

In models:

```ruby
class AnalyticsRecord < ActiveRecord::Base
  self.abstract_class = true
  connects_to database: { writing: :analytics, reading: :analytics
end
```

In config/database.yml:

```yaml
analytics:
  <<: *default
  migrations_paths: db/analytics_migrate
  database: storage/<%= Rails.env %>-analytics.sqlite3
```

PROBLEM: LONG TRANSACTIONS IN DATA MIGRATIONS

Rails wraps data migrations (change, up, down) in transactions by default:

```ruby
class MyMigration < ActiveRecord::Migration[7.1]
  def change
    # Transaction starts (IMMEDIATE lock)
    User.find_each do |user|
      # Potentially slow computation per user
      new_status = compute_new_status(user)
      user.update_columns(status: new_status) # Quick write
    end
    # Transaction ends (lock released)
  end
end
```

PROBLEM: LONG TRANSACTIONS IN DATA MIGRATIONS

Rails wraps data migrations (change, up, down) in transactions by default:

```ruby
class MyMigration < ActiveRecord::Migration[7.1]
  def change
    # Transaction starts (IMMEDIATE lock)
    User.find_each do |user|
      # Potentially slow computation per user
      new_status = compute_new_status(user)
      user.update_columns(status: new_status) # Quick write
    end
    # Transaction ends (lock released)
  end
end
```

PROBLEM: LONG TRANSACTIONS IN DATA MIGRATIONS

Rails wraps data migrations (change, up, down) in transactions by default:

```ruby
class MyMigration < ActiveRecord::Migration[7.1]
  def change
    # Transaction starts (IMMEDIATE lock)
    User.find_each do |user|
      # Potentially slow computation per user
      new_status = compute_new_status(user)
      user.update_columns(status: new_status) # Quick write
    end
    # Transaction ends (lock released)
  end
end
```

PROBLEM: LONG TRANSACTIONS IN DATA MIGRATIONS

Rails wraps data migrations (change, up, down) in transactions by default:

```ruby
class MyMigration < ActiveRecord::Migration[7.1]
  def change
    # Transaction starts (IMMEDIATE lock)
    User.find_each do |user|
      # Potentially slow computation per user
      new_status = compute_new_status(user)
      user.update_columns(status: new_status) # Quick write
    end
    # Transaction ends (lock released)
  end
end
```

SOLUTION: DATA MIGRATIONS

SOLUTION: DATA MIGRATIONS

1. Use Raw SQL: Often much faster than iterating with ActiveRecord models. Might be fast enough.

SOLUTION: DATA MIGRATIONS

1. Use Raw SQL: Often much faster than iterating with ActiveRecord models. Might be fast enough.

```
execute("UPDATE users SET status = 'new_status' WHERE ...")
```

SOLUTION: DATA MIGRATIONS

1. Use Raw SQL: Often much faster than iterating with ActiveRecord models. Might be fast enough.

```
execute("UPDATE users SET status = 'new_status' WHERE ...")
```

2. Disable Transaction: Add disable_ddl_transaction! to the migration class.

SOLUTION: DATA MIGRATIONS

1. Use Raw SQL: Often much faster than iterating with ActiveRecord models. Might be fast enough.

```
execute("UPDATE users SET status = 'new_status' WHERE ...")
```

2. Disable Transaction: Add disable_ddl_transaction! to the migration class.
3. Do complex data changes outside of deployment (e.g., in a Rake task, or a Job).

TRADE-OFF: FOREIGN KEY ERRORS

PostgreSQL: Gives detailed foreign key violation errors.

```
-- PG Example
ERROR: insert or update on table "orders" violates
       foreign key constraint "orders_user_id_fkey"
DETAIL: Key (user_id)=(999999) is not present in table "users".
```

You know exactly *which* constraint and *which* table failed.

TRADE-OFF: FOREIGN KEY ERRORS

SQLite: Errors are... less helpful.

```
-- SQLite Example (INSERT or DELETE)
Runtime error: FOREIGN KEY constraint failed (19)
```

DEBUGGING FK ERRORS

1. Disable foreign keys with PRAGMA `foreign_keys = OFF`
2. Execute query that breaks foreign keys
3. Run PRAGMA `foreign_key_check` which iterates over all foreign keys and finds broken ones

```
sqlite> PRAGMA foreign_keys = OFF;
sqlite> DELETE FROM users WHERE id = 1;
sqlite> PRAGMA foreign_key_check('orders');
orders|2|users|0
```

LIMITATION: SINGLE NODE BY DEFAULT

LIMITATION: SINGLE NODE BY DEFAULT

- SQLite database = just a file on disk.

LIMITATION: SINGLE NODE BY DEFAULT

- SQLite database = just a file on disk.
- Reads can be scaled with read replicas (using Litestream, etc.), but writes are limited to one node.

LIMITATION: SINGLE NODE BY DEFAULT

- SQLite database = just a file on disk.
- Reads can be scaled with read replicas (using Litestream, etc.), but writes are limited to one node.
- So, how do you scale?

SCALING OPTION 1: GO VERTICAL!

SCALING OPTION 1: GO VERTICAL!

- Modern servers are powerful.

SCALING OPTION 1: GO VERTICAL!

- Modern servers are powerful.
- SQLite is incredibly efficient.

SCALING OPTION 1: GO VERTICAL!

- Modern servers are powerful.
- SQLite is incredibly efficient.
- You can get a lot of performance from a single, large instance.

SCALING OPTION 1: GO VERTICAL!

- Modern servers are powerful.
- SQLite is incredibly efficient.
- You can get a lot of performance from a single, large instance.
- Example (Hetzner Cloud/Dedicated):
  - Our current: 16 vCPU (~$70/mo)
  - Available: 48 dedicated vCPU / 192GB RAM (~$330/mo)
  - Even larger bare metal options exist (256vCPU).

SCALING OPTION 2: REPLICATION

SCALING OPTION 2: REPLICATION

- Litestream, LiteFS

SCALING OPTION 2: REPLICATION

- Litestream, LiteFS
- Beamer (Rails World talk, not yet available)

SCALING OPTION 2: REPLICATION

- Litestream, LiteFS
- Beamer (Rails World talk, not yet available)
- libSQL with embedded replicas (libsql-ruby gem)

SCALING OPTION 2: REPLICATION

- Litestream, LiteFS
- Beamer (Rails World talk, not yet available)
- libSQL with embedded replicas (libsql-ruby gem)
- Turso DB (adds MVVC, for now experimental)

BACKUPS: LITESTREAM

BACKUPS: LITESTREAM

- Litestream continuously streams SQLite changes to S3-compatible storage (we use Cloudflare R2)

BACKUPS: LITESTREAM

- Litestream continuously streams SQLite changes to S3-compatible storage (we use Cloudflare R2)
- Point-in-time recovery: you can restore to any second within the specified period

BACKUPS: LITESTREAM

- Litestream continuously streams SQLite changes to S3-compatible storage (we use Cloudflare R2)
- Point-in-time recovery: you can restore to any second within the specified period
- Extremely cheap: if your DB weighs a few GBs, Cloudflare's free tier will likely cover it

BACKUPS: MANUAL FULL DB SNAPSHOTS

Install rclone, add your bucket and run this in cron:

```
sqlite3 "$db_file" ".backup $backup_file"
gzip "$backup_file"
rclone move "$backup_file.gz" "R2:$bucket/$backup_path/"
```

BACKUPS: MANUAL FULL DB SNAPSHOTS

Install rclone, add your bucket and run this in cron:

```
sqlite3 "$db_file" ".backup $backup_file"
gzip "$backup_file"
rclone move "$backup_file.gz" "R2:$bucket/$backup_path/"
```

BACKUPS: MANUAL FULL DB SNAPSHOTS

Install rclone, add your bucket and run this in cron:

```
sqlite3 "$db_file" ".backup $backup_file"
gzip "$backup_file"
rclone move "$backup_file.gz" "R2:$bucket/$backup_path/"
```

CONCLUSION: OUR JOURNEY & TAKEAWAYS

- **Why:** Driven by simplicity, cost, and team constraints.
- **Results:** Faster app, slashed costs, reduced DevOps overhead.
- **SQLite in Prod:** Viable & beneficial for many Rails apps
- **Is it for you?** Consider the trade-offs (locking, FK errors, scaling model) against the benefits.

QUESTIONS?

Wojtek Wrona @wojtodzio
X, Bluesky, LinkedIn

Code/Types Repo:
github.com/wojtodzio/activerecord-sqlite-types

THANK YOU!

Wojtek Wrona @wojtodzio
X, Bluesky, LinkedIn

Code/Types Repo:
github.com/wojtodzio/activerecord-sqlite-types